UNITED STATES PATENT APPLICATION FOR

ARCHITECTURE, SYSTEM, AND METHOD FOR OPERATING ON ENCRYPTED AND/OR HIDDEN INFORMATION

Inventors:

Richard C. Johnson

Andrew Morgan

H. Peter Anvin

Linus Torvalds

Prepared by:

WAGNER, MURABITO & HAO LLP
Two North Market Street
Third Floor
San Jose, California 95113
(408) 938-9060

FIELD OF THE INVENTION

The present invention generally relates to the field of memory architectures, hardware and software, and methods for operating on, manipulating and/or processing encrypted and/or hidden information (e.g., software code and/or data). More specifically, embodiments of the present invention pertain to a processor memory architecture and method for interpreting, translating, performing operations on and/or executing hidden and/or encrypted code and/or data.

SUMMARY OF THE INVENTION

Embodiments of the present invention relate to a processor memory architecture, system and method for operating on encrypted and/or hidden information. (In general, the phrases "operating on" and/or "performing [an] operation[s] on" refer to standard and/or conventional methods of operating on, manipulating, executing and/or processing code or data, and the term "encrypted" refers to standard and/or conventional techniques and/or software programs for encrypting, ciphering, transmuting, translating, converting and/or scrambling information from a plain text or conventionally recognizable graphics or other common form into a form designed, adapted or configured to prevent, deter, hinder or inhibit interception, deciphering, reading, comprehension and/or understanding by unauthorized parties.) In one embodiment of the present invention, the architecture enables storing encrypted information (e.g., software code and/or data) in unprotected (e.g., user-accessible) memory, while operating on an unencrypted

form of this information only in protected (e.g., user-inaccessible) memory. In a further exemplary embodiment, the present invention provides a scheme for efficiently hiding proprietary information in a system, architecture and/or process that converts information from a non-proprietary format or ISA into a proprietary format or ISA.

The present invention advantageously allows manufacturers of peripheral devices, electronic systems, etc. (e.g., original equipment manufacturers, or OEMs), to extend their shipping systems with encrypted code (e.g., x86-compatible instructions), and/or extend their systems with hidden proprietary intellectual property (e.g., binary code, data, etc.). Some OEMs, for example, have x86-compatible software that has value as a trade secret. In some cases, the OEMs embed their algorithm(s) in a custom chip (e.g., an ASIC, a ROM, flash memory, etc.). It is therefore desirable to keep proprietary code hidden from unprotected (e.g., x86) memory space.

Another advantage of the invention includes providing an ability to perform operations on hidden proprietary information, so that it is not visible to other non-proprietary (e.g., x86-compatible) processes in plain text form during normal system operations, and so that it does not appear in plain text form in unprotected or accessible (e.g., x86) memory / memory space or on disk. These and other advantages of the present invention will become readily apparent from the detailed description of preferred embodiments below.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 is a diagram showing an exemplary architecture and/or system implementing the present invention.

Figure 2 is a diagram showing an exemplary memory architecture for the system of Figure 1.

Figure 3 is a diagram showing a further exemplary memory architecture implementing the present invention.

Figure 4 is an exemplary table for correlating identifiers (e.g., keys) with memory addresses of corresponding pages in the memory of Figure 2.

Figure 5 is a flow chart describing an exemplary process in accordance with the present invention for decrypting and operating on encrypted, proprietary information.

Figure 6 shows an alternative embodiment of the present invention.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

Reference will now be made in detail to the preferred embodiments of the invention, examples of which are illustrated in the accompanying drawings. While the invention will be described in conjunction with the preferred embodiments, it will be understood that they are not intended to limit the invention to these embodiments. On the contrary, the invention is intended to cover alternatives, modifications and equivalents, which may be included within the spirit and scope of the invention as defined by the appended claims. Furthermore, in the following detailed description of the present invention, numerous specific details are set forth in order to provide a thorough understanding of the present invention. However, it will be readily apparent to one skilled in the art that the present invention may be practiced without these specific details. In other instances, well-known methods, procedures, components, and circuits have not been described in detail so as not to unnecessarily obscure aspects of the present invention.

Some portions of the detailed descriptions which follow are presented in terms of procedures, logic blocks, processing, and other symbolic representations of operations on data bits within a computer, processor, controller and/or memory. These descriptions and representations are the means generally used by those skilled in data processing arts to effectively convey the substance of their work to others skilled in the art. A procedure, logic block, process, etc., is herein, and is generally, conceived to be a self-consistent sequence of steps or instructions leading to a desired

result. The steps include physical manipulations of physical quantities. Usually, though not necessarily, these quantities take the form of electrical, magnetic, optical, or quantum signals capable of being stored, transferred, combined, compared, and otherwise manipulated in a computer system. It has proven convenient at times, principally for reasons of common usage, to refer to these signals as bits, values, elements, symbols, characters, terms, numbers, or the like.

It should be borne in mind, however, that all of these and similar terms are associated with the appropriate physical quantities and are merely convenient labels applied to these quantities. Unless specifically stated otherwise as apparent from the following discussions, it is appreciated that throughout the present application, discussions utilizing terms such as "processing," "operating," "computing," "calculating," "determining," "displaying" or the like, refer to the action and processes of a computer system, or similar processing device (e.g., an electrical, optical, or quantum computing device) , that manipulates and transforms data represented as physical (e.g., electronic) quantities. The terms refer to actions and processes of the processing devices that manipulate or transform physical quantities within a computer system's component (e.g., registers, memories, other such information storage, transmission or display devices, etc.) into other data similarly represented as physical quantities within other components.

The present invention concerns an architecture, method and system for manipulating, hiding, executing and/or operating on hidden and/or encrypted information (e.g., code and/or data). In one aspect of the invention, the architecture comprises an unprotected memory space

configured to store encrypted information, a first protected memory space configured to store at least part of a set of operating system instructions, and a second protected memory space configured to store a plain text (decrypted) version of the encrypted information, wherein the operating system instructions in the first protected memory space operate on the decrypted information in the second protected memory space.

In another aspect of the invention, the system comprises a processor and the above memory architecture, where the processor is configured to execute the operating system instructions.

In a further aspect of the invention, the method comprises the steps of (a) transferring encrypted and/or hidden information to a first protected memory address inaccessible to a user-accessible software program, but accessible to an operating system instruction set; (b) decrypting the encrypted information to form a decrypted version thereof; and (c) storing the first protected memory address in a second protected memory address inaccessible to a user-accessible software program, but accessible to an operating system instruction set, where the second protected memory address is linked to an original location of the encrypted and/or hidden information.

The invention further relates to hardware and/or software implementations of the present architecture, method and system. The invention, in its various aspects, will be explained in greater detail below with regard to preferred embodiments.

Figure 1 shows an exemplary hardware architecture 10 in which the present invention may be implemented. Processor 11 (which may include a microprocessor, microcontroller, ASIC, FPGA, CPLD, system on a chip or other logic circuit, integrated circuit or combination thereof) is generally configured to execute conventional software instructions, operate on data stored in memory 14, or perform other logic operations or processes consistent with such circuits and/or devices. Memory 14 is communicatively coupled to processor 11, and is configured to store code

(e.g., software program instructions) and/or data. Memory 14 may include random access memory (e.g., DRAM and/or SRAM), cache memory (which may be, e.g., L1 and/or L2), an optical data storage medium (such as a CD-ROM), a floppy disk, a conventional hard drive (which may be, e.g., a personal computer hard drive, a USB-compatible or other external hard disk drive, etc.), a detachable electronically erasable and programmable memory (such as a memory stick), etc. ROM 12, which may comprise one or more discrete memory devices, is communicatively coupled to processor 11, and is configured to store information in a generally non-erasable form (e.g., a BIOS). ROM 12, which preferably comprises a boot memory and preferably is generally inaccessible to the user, may include an EPROM, EEPROM, flash memory, etc. More preferably, ROM 12 comprises one or more conventional flash ROMs.

Still referring to Figure 1, bridge 16 conveys instructions and/or information from processor 11, memory 14 and/or ROM 12 to internal and/or external devices such as I/O device 18, PCI device 20 and/or other peripheral device 22. In a preferred embodiment, bridge 16 is a conventional south bridge apparatus, IC, device or circuit. Peripheral device 22 may comprise a storage medium, a printer, a display monitor, a Universal Serial Bus (USB) peripheral device, etc. The system and/or architecture of Figure 1 may comprise one or more I/O device 18 and/or one or more peripheral computer interface (PCI) device 20. There may be any number of peripheral devices 22 (which is optional) that is supportable by the system and/or architecture. Further examples of suitable systems and/or architectures to which the present invention is generally applicable are described in, e.g., U.S. Patent Nos. 6,031,992, 6,011,908, 6,175,896, 6,199,152, 6,327,660, 6,401,208, 6,408,376 and 6,415,379, and in "The Technology Behind CRUSOE™ Processors," by A. Klaiber, Transmeta Corp. (Jan. 2000) (available from the World Wide Web at www.transmeta.com), the relevant portions of which are each hereby incorporated by reference.

In certain preferred embodiments, the processor 11 executes software that translates and/or interprets instructions and/or code (hereinafter, "code") from a first conventional and/or non-

proprietary instruction set architecture ("ISA") or format (e.g., x86 or CISC, RISC, SPARC, etc.) into a second, optionally proprietary, ISA or format (e.g., a Very Long Instruction Word format employed in certain CRUSOE® processors, commercially available from Transmeta Corporation, Santa Clara, California). One conventional tool for such instruction interpretation and translation is CODE MORPHING™ software, or CMS™, tool and/or architecture (also available from Transmeta Corporation, Santa Clara, California on certain of its CRUSOE® processors). Consequently, a further aspect of the invention relates to a software tool configured to manage memory resources in system / architecture 10, preferably one having an ability to programmably allocate and/or partition memory resources to various different functions (e.g., unprotected/accessible memory, protected memory in which information is hidden to user-accessible programs and/or object code, private memory as described herein, etc.) within memory space 100 in system / architecture 10, such as that in the CMS tool. Thus, the size of each of the various memories within memory space 100 may be programmable and may be changed according to changes in design or performance criteria. The present invention provides particular advantage in systems and/or architectures employing instruction and/or code interpretation and/or translation and/or memory management.

Figure 2 shows a memory architecture (or memory space allocation diagram) 100 suitable for use with the present invention. Memory space 100 may comprise all or part of memory 14 and/or ROM 12, but preferably, comprises part of memory 14. Accessible and/or unprotected memory 102 begins at address 0 and continues through address M kb (where M may be any positive integer value; e.g., from 1 to 2^n, n being any positive integer, preferably ≥ 10, more preferably ≥ 13, even more preferably ≥ 15). Inaccessible and/or protected memory 104 begins at address L kb and continues through address 0 (where L may be any negative integer value; e.g., from 1 to -2^x, x being any positive integer, preferably ≥ 10, more preferably ≥ 12, even more preferably ≥ 13). Alternatively, protected memory 104 may be in a second memory address space, beginning at address 0 and continuing through address L kb (where L may be any positive integer value; e.g., from 1 to 2^x,

where x is as described above, and in which case unprotected memory 102 begins at address L+1 and continues through address L+M kb). Optionally, when the system or architecture employs instruction and/or code interpretation and/or translation, part of protected memory 104 may comprise an interpretation cache 106 for storing code and/or instructions for interpreting and, optionally, (re)compiling ISA code and storing interpreted and/or (re)compiled ISA code. In such a case, the remainder 108 of protected memory may comprise a translation cache, beginning at address K kb and continuing through address L kb (where K may be any negative integer value; e.g., from 1 to -2^y, y being any positive integer, preferably ≥ 9, more preferably ≥ 10).

In general, encrypted and or proprietary information may be stored in any part of memory 100. Preferably, however, proprietary information is stored in unencrypted form only in protected memory 104. Also, preferably, encrypted information in unprotected memory 102 is unencrypted, then stored in protected memory 104 prior to execution and/or operation on it by processor 11.

In some cases, it is more desirable to hide proprietary data than to hide the code that operates on such data. The present invention advantageously provides a mechanism for hiding proprietary data in protected memory (e.g., memory the addresses of which are protected [preferably by hardware] from user and/or software access), so that it is not visible either (a) in plain text or recognizable form from unprotected memory, or (b) to non-proprietary, conventional and/or untranslated software-controlled processes. The proprietary data may be stored in a protected buffer (e.g., register bank having an address requiring authorization or authentication before access thereto is allowed), and it may be accessed through instructions to load the proprietary information from (or store it to) a form of protected memory, identified herein as "private" memory. (In the present application, "memory" also refers to addressable memory space, regardless of the physical form such memory space takes.) In general, only authorized processes may execute instructions to load or store information (preferably encrypted and/or proprietary) in or from private memory. In one embodiment, an unauthorized process (e.g., instruction execution) will perform a "no operation," or

NOP, instruction in place of any unauthorized load or store operation that attempts to access private memory.

Figure 3 shows an exemplary implementation of the present invention in memory 100. Within protected memory 104 is private memory 200. Private memory 200 is configured to store unencrypted and/or plain text information (e.g., code and/or data) in a manner that protects or hides it from being observable in user-accessible or unprotected memory. Preferably, the system and/or architecture 10 further includes a table configured to associate encrypted information (particularly encrypted code) with the corresponding unencrypted information stored in protected memory 104 or private memory 200. In one embodiment, the table includes (1) an authorization key or message digest which corresponds to and/or is generated from information (code and/or data) in the encrypted code and/or data stored in unprotected memory, and (2) a pointer to the address in protected, private memory 206 where the unencrypted information is stored (see, e.g., Figure 4).

Referring back to Fig. 3, encrypted information may be decrypted and stored in protected memory 104 (and preferably private memory 200) shortly after system boot-up. When operating on encrypted data using, for example, authorized instructions to load and/or store the data (e.g., sometimes referred to "load x" and/or "store x," or LDX and/or STX), the table holds a list of pages that are authorized to execute such instructions, and therefore, that are authorized to access the corresponding plaintext copy of the encrypted data. Thus, each time the processor attempts to execute an instruction requiring authorization (e.g., load and/or store data that has been encrypted), the system checks the table for authorization. However, in the case of executing encrypted code, the table is accessed (and thus authorization is sought and/or confirmed) whenever execution enters a new page (of the code).

In one embodiment, the invention further includes new "load x" and "store x" instructions that proved authorized access to encrypted and/or proprietary information. These "load x" and

"store x" instructions read from and write to a plaintext copy of proprietary (or otherwise decrypted) information stored in a protected memory buffer. "Load x" and "store x" instructions can execute only in an authorized process; unauthorized execution will perform a "no operation," or NOP, instruction in place of an unauthorized "load x" or "store x" operation that attempts to access protected memory where the plaintext copy is stored (preferably, in private memory 200).

Referring to both Figures 3 and 4, a further exemplary implementation of how the present invention can be used will be illustrated using encrypted, proprietary data as an example. However, as explained above, the present invention is equally applicable to proprietary and/or encrypted code and/or encrypted, non-proprietary data as well.

As shown in Figure 4, an authorization key 302 may be selected and stored (preferably by the owner, controller or creator of the proprietary data) in a configuration table 300. The encrypted information (including the authorization key or corresponding message digest from the encrypted page) is preferably stored in unprotected memory 102 (preferably on a hard disk) by the system / architecture 10 manufacturer or user. (Hereinafter, the phrases "authorization key" and "message digest" are used somewhat interchangeably, except where the context of such use clearly distinguishes between the two phrases.) The configuration table 300 (or "trusted page" list of which table 300 is at least a part) may also be stored by the system / architecture 10 manufacturer or user in any of memory 14, ROM 12 or memory 100, but is preferably stored in a protected memory on ROM 12, by the system / architecture manufacturer. The data may be encrypted and unencrypted using any appropriate proprietary or conventional encryption/decryption program (such as DES, AES or PGP, but preferably DES) and the decryption (or "root") key. Where the message digest field is used as an authorization key (e.g, message digest 306 in table 300) to verify that a page of code is authorized to access encrypted data, the corresponding indicator field 308 and pointer field 310 are generally left unused because the operating system instructions (or other set of instructions configured to operate on data and/or code in protected memory, such as translation

and/or interpretation code) simply verifies that "load x" (e.g., LDX) and/or "store x" (e.g., STX) operations are executing from one or more pages whose message digest(s) appear in table 300.

At the time of manufacture, an OEM generally encrypts the table 300 and stores it in a ROM (e.g., ROM 12 in Fig. 1) along with a decryption key and, e.g., a message digest or authorization key. The ROM is generally protected from access by end users, but not from access by the system designer and/or manufacturer.

After booting the system, an unprotected program (e.g., including a driver in unprotected memory 102) loads the encrypted data file 202 (or page[s] thereof) into protected memory 106 using an MSR interface (i.e., standard "Model Specific Registers," which may be used to extend an x86 architecture as is known in the art). Alternatively, instructions may be added to the operating system for performing this function, or one or more virtual devices may be added to the north bridge of system / architecture 10 (not shown) to perform this function.

Operating system software (e.g., for interpreting and/or translating information in unprotected memory, such as the CMS tool from Transmeta Corp.) checks the encrypted data to determine whether the message digest matches a message digest in the configuration table(s). If so, decryption software (which may be stored in unprotected or protected memory) decrypts the data using the decryption key stored in protected memory or on ROM by the OEM, and stores the plain text version in a buffer 206 allocated in private memory 200. Alternatively, the configuration table and/or trusted page list can be encrypted, stored on disk and brought into protected memory 104 after boot in the same manner described above for protected proprietary data. Authorized "load x" (e.g., LDX) and/or "store x" (e.g., STX) instructions (e.g., instruction 204 stored within file 202) access the buffer 206. Unauthorized LDX/STX instructions (i.e., an LDX/STX instruction in a file for which no key is present in a configuration table and/or trusted page list) perform NOPs instead of LDX/STX instructions.

In the simplest implementation, pages of programs containing authorized "load x" and/or "store x" instructions are authorized using a one-way hash (e.g., SHA-1 hashing). However, any unique identifier (within statistical significance) for the page, or even the entire program, may be used to authorize the page and/or program. Although the term "page" is used extensively herein to refer to a unit of memory of known size and configuration (e.g., depth and width), any similar term or configuration of memory may be applied to the present invention (e.g., block, number of bytes, packet, etc.). Code from one (or more, but preferably just one page) of such program pages is compiled, hashed and linked so that it loads at a fixed or known address in private memory 200. In one example, the page size is 4 kb, and the SHA-1 hash is about 160 bits long. However, within design criteria and/or constraints (e.g., the size and allocations of ROM memory), the invention is implementable with any memory unit or page size (but preferably $\geq$ 1 kb and $\leq$ 1 Mb) and any hash length that is sufficiently long to render it infeasible for another page to hash to the same value (and that is, e.g., $\geq$ 16 bits and $\leq$ 100 bytes long).

Alternatively, one may hash the first X bytes of an encrypted page (where X is a number of from 1 to 2^q, preferably from 2^m to 2^n, where q is an integer of from 5 to 10, m is an integer of from 2 to 4, and n is an integer of from 4 to 8) and determine whether that partial hash matches the corresponding portion of any of the keys in table 300 (in which case table 300 should also include a field for storing partial page hashes). If so, the hash for the entire encrypted page may be taken, and key matching performed as described above. This technique may reduce the overhead associated with full hashing of entire pages of encrypted information. Also, the hash for the entire proprietary program may be taken and stored to authorize the entire program, but one may trade off higher key granularity (preferable where only a certain subset of pages of the program are used extensively) for fewer hashing and decrypting operations. Furthermore, one may take a hash for one or more pages of proprietary information within a proprietary program may function as a key for the entire program, but this is less preferred due to the potential for uncertainty among various versions of the same program.

Preferably, the pages containing LDX/STX operations are read-only, are not modified at load time, and are stored in unprotected memory at a fixed or known address. However, it is certainly within the scope of the invention to encompass applications including self-modifying code, updated code and/or code that may be moved or transferred from a first address to a second (or further) address according to application criteria.

In this embodiment, the pages to be authorized have their SHA-1 hash recorded at manufacturing. The SHA-1 hashes are saved on disk and encrypted using the key. At boot, the SHA-1 hashes are passed to decrypting, interpreting and/or translating software, decrypted, and stored in protected memory (preferably in ROM 12). The operating system (e.g., interpreter/translator) software verifies that any instance of LDX/STX is on a page the SHA-1 hash of which appears in the authorized page list in protected memory.

To hide proprietary code, the present invention is adapted to support direct execution of encrypted code from unprotected memory space. Code pages to be stored in unprotected memory space 102 will be encrypted using an appropriate key at the time of manufacturing. As described above, in one embodiment, encrypted pages will be identified by a list of their SHA-1 hashes (see Figure 4). Alternatively or additionally, a list may be created and stored in memory 14 (preferably ROM or a hard drive within memory 14) that identifies pages in unprotected memory 102 that are authorized to execute a load and/or store operation in protected memory 106 and/or private memory 200. These hashes can be encrypted and saved on disk at time of system / architecture 10 manufacturing, and in such a case, they will be loaded into protected memory after boot.

During execution, when processor 11 fetches an encrypted page (or system 10 "enters" the page) from unprotected memory 102, its SHA-1 hash will be looked up in table 300. If the hash 302 is found, but the decrypted information pointer 304 (i.e., the address/location in protected and/or private memory where the decrypted information can be found) is vacant, the page is

decrypted, its plain text version is stored in protected and/or private memory, and the SHA-1 entry is marked present and associated with the plain text page at its address in protected memory. Execution (e.g., interpretation, translation and/or other operations) continues on the plain text copy stored in protected and/or private memory. (In a preferred embodiment, decrypted information is stored in private memory 200, then the operating system interprets and translates the decrypted information into a proprietary format or ISA and stores the interpreted and translated information in translation cache 108, before the information is further operated on.) Consequently, two sets of interpreting software may be necessary (one for interpreting code in unprotected memory, the other for interpreting code in protected memory).

As is known in the art and as described above, one may do partial or full hashing. For example, one may hash the first X bytes of code and/or data in an encrypted page in unprotected memory 102 (where X is as described above; in one embodiment, X is 16). If the partial hash matches a partial-hash entry in table 300, the remainder of the encrypted page can be fully hashed, and if the full hash matches a full-hash entry in table 300, the encrypted page can be loaded into protected memory, decrypted, and operated on. Partial hashing may require an appropriate set of software tools for implementation; e.g., in addition to the above, a tool that prohibits instructions that cross page boundaries (e.g., to avoid mixing encrypted and plain text code or data), etc.

Alternatively, one may simply store the hash of the encrypted page in a known location or encrypt a portion of a page (e.g., substantially all of the page except for its [unencrypted] hash), rather than take the hash of each page of encrypted information each time it is accessed. For example, the SHA-1 hash for a 4 Kbytes page is 20 bytes long. If the first 20 bytes of the page is its SHA-1 hash, and the remaining (4K - 20) bytes are encrypted, then the hashing operation can be avoided when the page is accessed. One may simply try to match the first 20 bytes of a 4K page with entries in the table. If no match is found, the page does not contain encrypted code or data. If a match is found, to avoid potential errors such as false positives, one should still hash the

remaining (4K - 20) bytes and verify that the hash matches the first 20 bytes in the page. This alternative embodiment may also require appropriate software rules, such as a rule prohibiting instructions across page boundaries.

Advantageously, however, operating system software (and preferably, translation and/or interpretation code) rules further include, when operating on encrypted and/or hidden code, (i) disabling instruction breakpoints and (ii) disabling execution of single steps. These rules render single stepping through encrypted information impossible, thereby further inhibiting software (and certain types of hardware) accesses to the encrypted information.

In addition, one may employ a page locating mechanism conceptually similar to the hashing scheme described above. When an encrypted page authorized to perform LDX/STX instructions is accessed (e.g., in the TLB miss path or other point in the system memory hierarchy when a page access is performed) for the first time after boot up, rather than hashing the encrypted page, the encrypted page is simply decrypted and stored in private memory 200. The address of the decrypted (plain text) page in private memory 200 is stored in the "Pointer" column of table 300, and is linked to the address of the encrypted page in unprotected memory 102 stored in the "SHA-1" column of table 300, so that the operating system of system/architecture 10 can locate and operate on the plain text code and/or data in protected memory. (In such an embodiment, the "SHA-1" column would be appropriately relabeled "Encrypted Information Address," "Disk Address," etc.) Thereafter, in further accesses of the authorized encrypted page, the operating system looks up the address(es) in table 300, then operates directly on the decrypted/hidden information stored in protected memory identified in table 300. This embodiment retains the essence of the other embodiments described above, in which a unique identifier links, or is otherwise associated with, both encrypted information in unprotected memory and a location or address in protected memory for the corresponding decrypted and/or hidden information.

The present invention also enables combining hidden code and data, so that hidden data can only be accessed from encrypted pages.

Constraints on the present scheme include a maximum size of encrypted pages that must fit in protected memory 106. A typical upper-bound on this size is 16MB, but of course, this size can be adjusted or even programmed, according to design. Furthermore, encrypted pages should be read-only and should not be modified during load operations. Furthermore, encrypted code and data should always load at the same linear address. In addition, encrypted pages of code preferably should not contain data, since there may be inefficiencies in existing software tools to redirect a load operation from an encrypted page to the plain text copy stored in protected memory.

Figure 5 is a flow chart showing exemplary steps in the process/method of the present invention. Process 400 comprises a number of steps 410-480. Initially, in step 410, the proprietary information is encrypted, and a decryption key and configuration table are generated. Optionally, and as described above, the configuration table may also be encrypted, moved and stored in the same manner as for encrypted information. Typically, step 410 is performed by the manufacturer, owner, creator and/or distributor of the proprietary information, who typically transmits the encrypted proprietary information, decryption key and configuration table to the manufacturer, designer, creator and/or distributor of system / architecture 10 for inclusion therein.

Including and/or incorporating the encrypted proprietary information, the encrypted or unencrypted configuration table / trusted page list, and decryption key in system / architecture 10 generally comprise steps 415 and 420 of Figure 5. In step 415, the encrypted information and, if encrypted, the configuration table is stored in unprotected (accessible) memory 102. In step 420, the decryption key and, if unencrypted, the configuration table are stored in protected (inaccessible) memory 106 or private memory 200, but preferably in a ROM (more preferably a flash memory ROM, so that it can be deleted, replaced or overwritten if desired or necessary). At boot up, all

pointers in the table or list are designated zero or origin, and then are replaced with information designating the actual address or location in protected or private memory (preferably private memory) where the plain text information is found after decryption.

In step 425, system / architecture 10 determines whether it is to operate on the encrypted information stored in unprotected memory 102. If not, the operating system in system / architecture 10 in step 435 determines whether loading and/or storing instructions are being executed. If such instructions are not being executed (and/or are not in an instruction queue for execution), the architecture / system 10 and operating system continue normal processing in step 440, as encrypted information is not being operated on.

However, if load x and/or store x instructions are being executed (and/or are in a queue to be executed), the operating system then calculates the page hash for the encrypted page associated with the loading and/or storing instructions being executed. In step 450, that hash is compared with the hashes in a corresponding configuration table that stores authorized hashes for that encrypted information. If the calculated hash matches an authorized hash, in step 460, the load x / store x instruction is allowed to execute and to access the plaintext version of the encrypted data stored (in one embodiment) in protected buffer 206, and the system / architecture 10 and operating system continue normal processing in step 440. If the calculated hash does not match any authorized hash, the operating system executes a "no operation" instruction, effectively aborting the load/store instructions.

Referring back to step 425 in Figure 5, if the operating system determines that it is to operate on encrypted information, then in step 430, system / architecture 10 determines whether table 300 has stored therein a corresponding non-zero pointer (e.g., an address location in protected memory 104 and/or private memory 200 where the decrypted proprietary information corresponding to the encrypted information can be found). If such an "all zeros" pointer is found

in table 300, the decrypted proprietary information corresponding to the encrypted information is not yet stored in protected memory. Thus, in step 470, the operating system decrypts the encrypted proprietary information using conventional software tools for such operations. Generally, the information is decrypted using the decryption key stored in the manufacturer-supplied ROM (or, in an alternative embodiment, as a page-specific decryption key in table 300), the plain text or decrypted information is stored in protected and/or private memory, and the new, non-zero pointer that corresponds to the location of the decrypted information is also stored in table 300, prior to other operating system operations (e.g., interpretation and translation) on the plain text page in step 480. Referring back to step 430, if the pointer has a non-zero value, the decrypted proprietary information corresponding to the encrypted information is in protected memory, and the operating system can execute from the corresponding plain text page.

Decryption could happen as early as when encrypted information pages are first loaded into unprotected memory (e.g., by keeping a shadow plain text copy in protected and/or private memory), or as late as when bytes are fetched by an appropriate operating system tool (such as an interpreter or [pre]fetcher; this may be considered to be an "on demand" scheme). In one implementation, decryption is performed on demand when pages are first accessed by an operating system tool, instruction and/or operation (and in a preferred embodiment, the tool is an interpreter), and a cache of the corresponding plain text pages is kept in protected memory.

Although specific steps are disclosed in process 400 of FIG. 5, such steps are exemplary. That is, the present invention is well suited to use with various other steps or variations of the steps recited in process 100. Additionally, for purposes of clarity and brevity, the discussion is directed at times to specific examples. The present invention (e.g., system/architecture 10 and/or process 400), however, are not limited solely to use with a particular device (e.g., a CPU or microprocessor) or with particular code (e.g., an x86 instruction set). Instead, the present invention is well suited to use with other types of hardware and software in which it may be

desirable to accomplish a multitude of tasks as part of an overall process directed at parsing hidden code.

Figure 6 shows an alternative scheme for operating on encrypted and/or hidden information, so that the information is not in unprotected memory in a plain text (or decrypted) form. This alternative scheme relies on the well-known principle that any sequence of information, when exclusively OR'ed ("XORed") twice, gives back the original sequence of information.

In Figure 6, pages of proprietary information 502a, 502b and 502c are XORed with pages of random bits 504a, 504b and 504c. Pages of proprietary information 502a, 502b and 502c may be information for the same or different application (e.g., a software tool and/or peripheral device, etc.). Furthermore, pages of random bits 504a, 504b and 504c, which are generally different from each other, serve a purpose similar to the decryption key and/or hash in the above scheme, particularly in that the manufacturer, supplier and/or creator of the proprietary information must perform the first XOR operation to encrypt the proprietary information and must also provide the page of random bits for inclusion in or incorporation into the system/architecture 10. Also, each page of proprietary information 502x is generally associated with a unique page of random bits 504x.

The once-XORed, encrypted proprietary information 506a, 506b and/or 506c ("506x") may then be stored in unprotected memory 102 (preferably on disk 510), and there may be as many pages of encrypted proprietary information stored in unprotected memory 102 as may fit therein. A configuration table (similar to that shown in FIG. 4, but not shown in FIG. 6) may contain hashes (e.g., SHA-1 hashes) of the once-XORed pages 506a, 506b and/or 506c. Page(s) of random bits 504a, 504b and/or 504c ("504x") are generally stored in protected memory 104, and a pointer identifying the location of random bit pages 504a, 504b and/or 504c in protected memory 104 is generally stored in a corresponding field of the configuration table. In this

embodiment, table 300 (at least initially) contains at least (1) a unique identifier for the page of proprietary information 506x and (2) a corresponding pointer to the location where the appropriate page of random bits 504x is stored.

A once-XORed, encrypted page 506x is decrypted by a subsequent XOR operation with the corresponding random bit page 504x to obtain decrypted proprietary information 508x (which is identical to the original proprietary information 502x). To avoid any need to store a plain text version of the encrypted page, the second XOR operation may be done on demand by XORing the encrypted bytes with the associated bytes located at the same offset within the page of random bits 504x corresponding to the current encrypted page 506x. This decryption operation occurs before (preferably as soon as possible before) execution of decrypted proprietary information 508x in private memory 200 and/or protected memory 104. Thus, pages of random bits 504x serve as a decryption tool, much like the hash or decryption key described above.

CONCLUSION/SUMMARY

Thus, the present invention provides a convenient, simple and efficient architecture, method and system for operating on hidden and/or encrypted information. If and/or when combined with an effective technique for preventing and/or inhibiting physical access to decrypted and/or hidden code (e.g., by removing a ROM, cache memory, or microprocessor IC from its board, removing the cap / top packaging materials, delayering the device, and using known reverse engineering techniques to determine the states of various memory bits/circuits thereon), one may completely prevent or effectively inhibit access to hidden code and/or plain text versions of encrypted code and/or data (which, as described herein, is kept hidden from software access by the present method, system and memory architecture).

The foregoing descriptions of specific embodiments of the present invention have been presented for purposes of illustration and description. They are not intended to be exhaustive or to

limit the invention to the precise forms disclosed, and obviously many modifications and variations are possible in light of the above teaching. The embodiments were chosen and described in order to best explain the principles of the invention and its practical application, to thereby enable others skilled in the art to best utilize the invention and various embodiments with various modifications as are suited to the particular use contemplated. It is intended that the scope of the invention be defined by the Claims appended hereto and their equivalents.